MOOCHO Reference Manual

Version of the Day

Generated by Doxygen 1.4.2

Wed Sep 27 02:16:26 2006

Contents

1 MOOCHO: Multi-functional Object-Oriented arCHitecture for Optimization

WARNING! This documention is currently under active construction!

1.1 Outline

- Introduction
- Hyper-linked HTML version of this Document
- MOOCHO Quickstart
  - Configuring, Building, and Installing MOOCHO
  - Installed NLP Examples
- Browse all of MOOCHO as a single doxygen collection

1.2 Introduction

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems (i.e. nonlinear programs) using reduced-space successive quadratic programming (SQP) methods. The most general form of the optimization problem that can be solved is:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & c(x) = 0 \\ & x_L \leq x \leq x_U \end{array}$$

where $x \in \Re^n$ the optimization variables, $f(x) \in \Re^n \rightarrow \Re$ is the nonlinear scalar objective function, $c(x) = 0$ (where $c(x) \in \Re^n \rightarrow \Re^m$) are the nonlinear constraints, and x_L and x_U are the upper and lower bounds on the variables. The current algorithms in MOOCHO are well suited to solving optimization problems with massive numbers of unknown variables and equations but few so-called degrees of optimization freedom (i.e. the degrees of freedom = $n - m$ = the number of variables minus the number of equality constraints). Various line-search based globalization methods are available, including exact penalty functions and a form of the filter method. The algorithms in MOOCHO are provably locally and globally convergent for a wide class of problems in theory but in practice the behavior and the performance of the algorithms varies greatly from problem to problem.

MOOCHO was initially developed to solve general sparse optimization problems where there is no clear distinction between state variables and optimization parameters. For these types of problems a serial sparse direct solver must be used (only if you have MA28) to find a square basis for the variable reduction decompositions current supported.

More recently, MOOCHO has been interfaced through `Thyra` and the `Thyra::ModelEvaluator` to address very large-scale, massively parallel, simulation-constrained optimization problems that take the form:

$$\begin{array}{ll} \mbox{minimize} & f(y,u) \\ \mbox{subject to} & c(y,u) = 0 \\ & y_L \leq y \leq y_U \\ & u_L \leq u \leq u_U \end{array}$$

where $y \in \Re^{n_y}$ are the state variables, $u \in \Re^{n_u}$ are the optimization parameters and $c(y,u) = 0$ are the discrete nonlinear state simulation equations. Here the state Jacobian $\frac{\partial c}{\partial y}$ must be square and nonsingular and the partitioning of $x = \left[\begin{array}{cc} y^T & u^T \end{array} \right]^T$ into state variables y and optimization variables u must be known *a priori* and this partitioning can not change during a solve. All of the functionality needed for MOOCHO to solve a simulation-constrained optimization problem can be specified through sub-classing the `Thyra::ModelEvaluator` interface, and related `Thyra` interfaces. Epetra-based applications can instead implement the `EpetraExt::ModelEvaluator` interface and never need to work with Thyra directly except in trivial and transparent ways.

For simulation-constrained optimization problems, MOOCHO can utilize the full power of the massively parallel iterative linear solvers and preconditioners available in Trilinos through Thyra through the `Stratimikos` package by just flipping a few switches in a parameter list. These include all of the direct solves in Amesos, the preconditioners in Ifpack and ML, and the iterative Krylov solvers in AztecOO and Belos (which is not being released but is available in the development version of Trilinos). For small to moderate numbers of optimization parameters, the only bottleneck to parallel scalability is the linear solver used to solve linear systems involving the state Jacobian $\frac{\partial c}{\partial y}$. The reduced-space SQP algorithms in MOOCHO itself exhibit extremely good parallel scalability. The parallel scalability of the linear solvers is controlled by the simulation application and the Trilinos linear solvers and preconditioners themselves. Typically, the parallel scalability is limited by the preconditioners as the problem is partitioned to more and more processes.

MOOCHO also includes a minimally invasive mode for reduced-space SQP where the simulator application only needs to compute the objective and constraint functions

$f(y,u) \in \Re^{n_y+n_u} \rightarrow \Re$ and $c(y,u) \in \Re^{n_y+n_u} \rightarrow \Re^{n_y}$ and solve only forward linear systems involving $\frac{\partial c}{\partial y}$. All other derivatives can be approximated with directional finite differences but any exact derivatives that can be computed by the application are happily accepted and fully utilized by MOOCHO through the `Thyra::ModelEvaluator` interface.

A more detailed mathematical overview of nonlinear programming and the algorithms that MOOCHO implements are described in the document `An Overview of MOOCHO"`.

Note: Specific documentation on the algorithms that MOOCHO implements and the use of MOOCHO will be added shortly as well as detailed examples.

1.3 Hyper-linked HTML version of this Document

The doxygen-generated hyper-linked version of his document can be found at the Trilinos website under the link to MOOCHO.

1.4 MOOCHO Quickstart

In order to get started using MOOCHO to solve your NLPs you must first build MOOCHO as part of Trilinos and install it. When MOOCHO is installed with Trilinos, several complete examples are also installed that show how to define NLPs, compile and link against the installed headers and libraries, and how to run the MOOCHO solvers.

Below, we briefly describe Configuring, Building, and Installing MOOCHO, and accessing the Installed NLP Examples.

1.4.1 Configuring, Building, and Installing MOOCHO

Complete details on the configuration, building, and installing of Trilinos are described in the Trilinos Installation Guide. However, we give a quick overview of one such installation that works on Linux systems using g++.

Here we describe the configuration, build, and installation process for a directory structure that looks like:

```
  $TRILINOS_BASE_DIR
   |
   |-- Trilinos
   |
    -- BUILDS
        |
         -- DEBUG
```

where `$TRILINOS_BASE_DIR` is some base directory such as `TRILINOS_BASE_DIR=~/PROJECTS/Trilinos.base`. However, in general, the build directory (show as `$TRILINOS_BASE_DIR/BUILDS/DEBUG` above) can be any directory you want but should not be the same as the base directory for Trilinos. In the most general case, we will assume that `$TRILINOS_BUILD_DIR` is the base build directory; in this section, we assume that `TRILINOS_BUILD_DIR=$TRILINOS_BASE_DIR/BUILDS/DEBUG`.

Here are the steps needed to configure, build, and install MOOCHO along with the rest of Trilinos:

1. **Obtain an expanded source directory tree for Trilinos and create a build directory**

   Once you have created the directories `$TRILINOS_BASE_DIR` and `$TRILINOS_BASE_DIR/BUILDS/DEBUG` you need to get a copy of the Trilinos source.

   If you have CVS access you can obtain the version of the day through the main development trunk or can check out a specific tagged release. For example, to obtain the version of the day you would perform:

   ```
   cd $TRILINOS_BASE_DIR
   cvs -d :ext:userid@software.sandia.gov:/space/CVS co Trilinos
   ```

   where `userid` is your user ID on the CVS server. For further details on working with CVS access to Trilinos, see the `Trilinos Developers Guide`.

   If you do not have CVS access you can obtain an tar-bar for a release of Trilinos from the `Trilinos Releases Download Page`. Once you have the tar ball, you can expand it into the directory `$TRILINOS_BASE_DIR` as follows:

   ```
   cd $TRILINOS_BASE_DIR
   tar -xzvf ~/Trilnos.xxx
   ```

2. **Create a configuration script**

   By far the hardest part of building and installing Trilinos is figuring out how to write the configuration script. The best place to find example configure scripts that at least have a chance of being correct on specific systems is to look at Trilinos test harness scripts in the directory:

   ```
   Trilinos/commonTools/test/harness/invoke-configure
   ```

   Older scripts that have worked on a wider variety of systems in the past can be found in the directory:

```
  Trilinos/sampleScripts
```

Warning! The above scripts are likely to be currently broken for even the same systems for which they where developed. These scripts really only provide ideas for different combinations to try to get a configure script to work on your system.

Below is an example configure script called `do-configure` that might be used to configure Trilinos with MOOCHO support (but might not actually work on any actual computer on Earth):

```
$TRILINOS_BASE_DIR/Trilinos/configure \
--prefix=$TRILINOS_INSTALL_DIR \
--with-install="/usr/bin/install -p" \
--with-gnumake \
--enable-export-makefiles \
--with-cflags="-g -O0 -ansi -Wall" \
--with-cxxflags="-g -O0 -ansi -Wall -ftrapv -pedantic -Wconversion" \
--enable-mpi --with-mpi-compilers \
--with-incdirs="-I${HOME}/include" \
--with-ldflags="-L${HOME}/lib/LINUX_MPI" \
--with-libs="-ldscpack -lumfpack -lamd -lparmetis-3.1 -lmetis-4.0 -lskit" \
--with-blas=-lblas \
--with-lapack=-lapack \
--with-flibs="-lg2c" \
--disable-default-packages \
--enable-teuchos --enable-teuchos-extended --disable-teuchos-complex \
  --enable-teuchos-abc --enable-teuchos-debug \
--enable-thyra \
--enable-epetra \
--enable-triutils \
--enable-epetraext \
--enable-amesos --enable-amesos-umfpack --enable-amesos-dscpack \
--enable-aztecoo \
--enable-ifpack --enable-ifpack-metis --enable-ifpack-sparskit \
--enable-ml --with-ml_metis --with-ml_parmetis3x \
--enable-stratimikos \
--enable-moocho
```

The above script is almost completely platform dependent in most cases, except for everything below `-disable-default-packages` for enable options for individual packages. A few points about the above configure script are worth mentioning. First, some of the package enable options such as `-enable-epetra` should be unnecessary once other options such as `-enable-epetraext` are included but to be safe it is a good idea to be explicit about what packages to build in case the built in top-level configure logic is wrong. Second, it is a good idea to include the options `-enable-teuchos-debug` and `-enable-teuchos-abc` when you first start working with Trilinos to help catch coding errors on your part (and perhaps on the part of Trilinos developers). Third, the above script show enabled support for several third-party libraries such as UMFPACK, DSCPACK, SparseKit, and Metis. You are responsible for installing these third party libraries yourself if you want the extra capabilities that they enable. Otherwise, to get started, a simpler script such as the follow can be used to begin with:

```
$TRILINOS_BASE_DIR/Trilinos/configure \
--prefix=$TRILINOS_INSTALL_DIR \
--with-install="/usr/bin/install -p" \
--with-gnumake \
--enable-export-makefiles \
--with-cflags="-g -O0 -ansi -Wall" \
--with-cxxflags="-g -O0 -ansi -Wall -ftrapv -pedantic -Wconversion" \
--enable-mpi --with-mpi-compilers \
--with-blas=-lblas \
--with-lapack=-lapack \
--with-flibs="-lg2c" \
--disable-default-packages \
--enable-teuchos --enable-teuchos-extended --disable-teuchos-complex \
  --enable-teuchos-abc --enable-teuchos-debug \
--enable-thyra \
--enable-epetra \
--enable-triutils \
--enable-epetraext \
--enable-amesos \
--enable-aztecoo \
--enable-ifpack \
--enable-ml \
--enable-stratimikos \
--enable-moocho
```

As a final step, you can make the `do-configure` script executable and this is assumed below.

3. **Configure, build, and install Trilinos**

   Once you have a configure script, you can configure and build Trilinos as follows:

```
  cd $TRILINOS_BUILD_DIR
  ./do-configure
  make
  make install
```

   If a problem does occur, it usually occurs during configuration. Often trial and error is required to get the configuration to complete successfully.

   Once the Trilinos build completes (which can take hours on a slower machine if a lot of packages are enabled) you should test Trilinos using something like:

```
  make runtest-mpi TRILINOS_MPI_GO="mpirun -np "
```

If you not enable MPI, you run run the serial test suite as:

```
make runtest-serial
```

Some of the tests in the test suite may fail if you have not enabled everything and this is okay. Once you feel good about the build, you can install Trilinos as follows:

```
make install
```

If everything goes smoothly, then Trilinos will be installed with the following directory structure:

```
$TRILINOS_INSTALL_DIR
 |
 |-- examples
 |
 |-- include
 |
 |-- libs
 |
  -- tools
```

Once the install competes, you can move on to building and running the external MOOCHO examples as describe in the next section.

1.4.2 Installed NLP Examples

When the configure option `-enable-export-makefiles` is included, a set of examples are installed in the directory specified by `-prefix=$TRILINOS_INSTALL_DIR` and the directory structure will look something like:

```
$TRILINOS_INSTALL_DIR
 |
 |-- examples
 |    |
 |     -- moocho
 |         |
 |         |-- NLPWBCounterExample
 |         |
 |         |-- ExampleNLPBanded
 |         |
 |         |-- thyra
 |              |
 |              |-- NLPThyraEpetraModelEval4DOpt
 |              |
 |               -- NLPThyraEpetraAdvDiffReactOpt
 |
  -- tools
      |
       -- moocho
```

Note that the directory `$TRILINOS_INSTALL_DIR/examples/moocho/thyra` will not be installed if `-enable-export-makefiles` is not included (or is disabled) or if `-enable-thyra` is missing or `-disable-moocho-thyra` is specified at configure time.

Each installed example contains a simple makefile that is ready to build each of the examples and to demonstrate several important features of MOOCHO. Each makefile shows how to compile and link against the installed header files and libraries. These makefiles use the Trilinos export makefile system to make it easy to get all of the compiler and linker options and get the right libraries in the build process. The user is encouraged to copy these examples to their own directory and modify them to solve their NLPs.

Specific examples can be examined through the links below but we first go through the common features of these examples here for one of the `Thyra::ModelEvaluator examples`.

One common feature of all of these examples is the makefile that is generated. For the `NLPWBCounterExample` example (that is described ???here???) the makefile looks like:

By using the macros starting with `MOOCHO_` one is guaranteed that the same compiler with the same options are used to build the client's code that were used to build Trilinos. Of particular importance are `MOOCHO_CXX`, `MOOCHO_DEFS`, `MOOCHO_CPPFLAGS`, and `MOOCHO_CXXLD` since these ensure that the same C++ compiler and the same `-D` macro definitions are used. These are critical to compiling compatible code in many cases. The macros `MOOCHO_LIBS` contain all of the libraries needed to link executables and they include all of the libraries in their lower level dependent Trilinos packages. For example, you don't explicitly see the libraries for say Teuchos, but you can be user that they are there.

This makefile gets created with the following lines commented in or out depending on if `-enable-gnumake` was specified or not:

In the above example, support for GNU Make is enabled which results in scripts being called to clean up the list of include paths and libraries and have duplicate entries removed.

1.5 Browse all of MOOCHO as a single doxygen collection

You can browse all of MOOCHO as a `single doxygen collection`. Warning, this is not the recommended way to learn about MOOCHO software. However, this is a good way to browse the `directory structure of MOOCHO`, to <ahref="../../browser/doc/html/files.html">locate files, etc.